National University of Computer & Emerging Sciences (FAST-NU)

Operating System Project Report

By:

Dehya Khurraim : 20K-0128

Muhammad Huzaifa : 20K-0257

Tayyab Shahzad : 20K-1043

Contents

INTRODUCTION

The basic functionality of this PAGE REPLACEMENT project is that it implements page replacement algorithms we study in class.

METHODOLOGY

This software implements the page-replacement algorithms FIFO, Optimal, MFU, and LRU algorithms. There is given a text file contain string of page numbers (ranging from 0 to 9) create a page-reference string. Software applies the page-reference string to each algorithm and reports the number of page faults, number of hits & hit ratio each algorithm has encountered. Algorithms implemented are:

- FIFO: This is the most basic page replacement method. The operating system uses this algorithm to maintain track of all pages in memory in a queue, with the oldest page at the top. When a page needs to be replaced, the first page in the queue is chosen for replacement.

- Optimal: Pages that will not be used for the greatest period of time in the future are replaced using this technique.

- MFU: This approach is based on the assumption that the page with the lowest count was most likely freshly added and has yet to be used.

- LRU: This method will replace the page that has been used the least lately.

HOW PROJECT STARTS

The project is Linux based and the first step of the project is to have the latest version of ubuntu so that there are no errors in the working of the programs. All the codes will be written and executed separately on the terminal or if ubuntu allows all of the approaches will be on the same .c file.

CODE

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <stdbool.h>

#define MAX_LINE 1024

int array[MAX_LINE / 2];
int page_faults = 0;
int hit = 0;
int ratio;

void initialize_check(int working_set_size)
{
  page_faults = 0;
  hit = 0;
  int j = 0;
  for (j = 0; j < working_set_size; j++)
  {
    array[j] = 9999;
  }
}
```

```
int check_present(int check, int working_set_size)
{
  int present = 0;
  int k = 0;
  while (k < working_set_size)
  {
    if (array[k] == check)
    {
      present = 1;
      break;
    }
    k++;
  }
  return present;
}

bool page_found(int pages[], int working_set_size, int page_search, int *counter)
{
  int j = 0;
  for (j = 0; j < working_set_size; j++)
  {
    if (pages[j] == page_search)
    {
      *counter = j;
      return true;
    }
  }
  *counter = -1;
  return false;
}
```

```
int page_blank(int pages[], int working_set_size)
{
  int j = 0;
  for (j = 0; j < working_set_size; j++)
  {
    if (pages[j] == -1)
    {
      return 1;
    }
  }
  return 0;
}

void FIFO_pagefault(int pages[], int working_set_size, int length)
{
  initialize_check(working_set_size);
  int i, j = 0;
  for (i = 0; i <= length; i++)
  {
    if (check_present(pages[i], working_set_size) == 0)
    {
      for (j = 0; j < working_set_size - 1; j++)
      {
        array[j] = array[j + 1];
      }
      array[j] = pages[i];
      page_faults = page_faults + 1;
    }
```

```
else
    {
            hit++;
    }
  }
  ratio = (hit*100)/(length+1);
  printf("\n\t\t\t\tFIFO :\n\n");
  printf("No. of Hits : %2d\t",hit);
  printf("No. of Page Falut: %2d\t\t",page_faults);
  printf("Hit Ratio: %2d%%\n",ratio);
}

void optimal_pagefault(int pages[], int working_set_size, int length)
{
  int i, j, k = 0;
  int close[MAX_LINE / 2];

  initialize_check(working_set_size);
  while (k <= length)
  {
    if (check_present(pages[k], working_set_size) == 0)
    {
      for (i = 0; i < working_set_size; i++)
      {
        int find = 0;
        int page = array[i];
        j = k;
        while (j < length)
        {
```

```
if (page == pages[j])
        {
          find = 1;
          close[i] = j;
          break;
        }
        else
        {
          find = 0;
        }
        j++;
      }
      if (!find)
      {
        close[i] = 9999;
      }
    }
    int maximum = -9999;
    int repeated;
    i = 0;
    while (i < working_set_size)
    {
      if (maximum < close[i])
      {
        repeated = i;
        maximum = close[i];
      }
      i++;
    }
```

```
     array[repeated] = pages[k];
      page_faults = page_faults + 1;
    }
    else
    {
          hit++;
    }
    k++;
  }

  ratio = (hit*100)/(length+1);
  printf("\n\t\t\t\tOPTIMAL :\n\n");
  printf("No. of Hits : %2d\t",hit);
  printf("No. of Page Falut: %2d\t\t",page_faults);
  printf("Hit Ratio: %2d%%\n",ratio);
}

void LRU_pagefault(int pages[], int working_set_size, int length)
{
  int i, j, k = 0;
  int close[MAX_LINE / 2];
  initialize_check(working_set_size);
  while (k <= length)
  {
    if (check_present(pages[k], working_set_size) == 0)
    {
      for (i = 0; i < working_set_size; i++)
      {
        int find = 0;
        int page = array[i];
```

```
  j = k - 1;
  while (j >= 0)
  {
    if (page == pages[j])
    {
      find = 1;
      close[i] = j;
      break;
    }
    else
    {
      find = 0;
    }
    j--;
  }
  if (!find)
  {
    close[i] = -9999;
  }
}
int least = 9999;
int repeated;
i = 0;
while (i < working_set_size)
{
  if (close[i] < least)
  {
    repeated = i;
    least = close[i];
  }
```

```
      i++;
     }
     array[repeated] = pages[k];
     page_faults = page_faults + 1;
    }
    else
    {
          hit++;
    }
    k++;
  }
  ratio = (hit*100)/(length+1);
  printf("\n\t\t\t\tLRU :\n\n");
  printf("No. of Hits : %2d\t",hit);
  printf("No. of Page Falut: %2d\t\t",page_faults);
  printf("Hit Ratio: %2d%%\n",ratio);
}

void MFU_pagefault(int working_set_size, char copy_forMFU[])
{
  page_faults = 0;
  int pages[working_set_size];
  int array_copy[MAX_LINE];
  int pages_copy[working_set_size];
  int page_counter = 0;
  int prev_counter = 0;
  int final_counter = 0;
  int counter = 0;
```

```
int i = 0;
for (i = 0; i < MAX_LINE; i++)
{
  array_copy[i] = -1;
}
for (i = 0; i < working_set_size; i++)
{
  pages_copy[i] = 0;
  pages[i] = -1;
}

int length = 0;
char *token = strtok(copy_forMFU, " ");

while (token != NULL)
{
  array_copy[length] = atoi(token);
  token = strtok(NULL, " ");
  length++;
}

int check_pages = page_blank(pages, working_set_size);

while (check_pages == 1)
{
  if (!page_found(pages, working_set_size, array_copy[page_counter], &counter))
  {
    pages[prev_counter] = array_copy[page_counter];
    pages_copy[prev_counter]++;
    page_faults++;
```

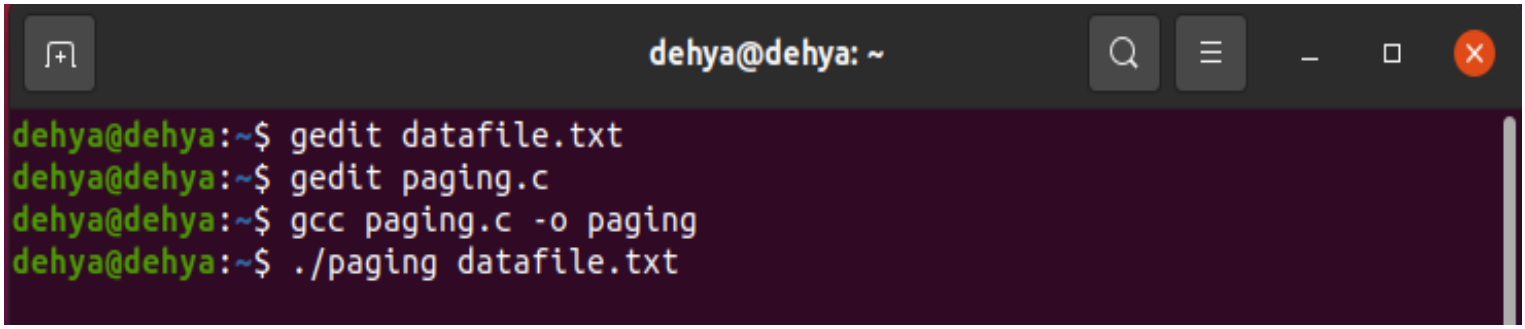
```
    prev_counter++;
  }
  else if (page_found(pages, working_set_size, array_copy[page_counter], &counter))
  {
    pages_copy[counter]++;
  }
  page_counter++;
  check_pages = page_blank(pages, working_set_size);
}



for (final_counter = page_counter; final_counter <= length; final_counter++)
{
  if (page_found(pages, working_set_size, array_copy[final_counter], &counter))
  {
    pages_copy[counter]++;
    continue;
  }

  int max_occur = pages_copy[0];
  int k = 0;
  counter = k;
  for (k = 1; k < working_set_size; k++)
  {
    if (max_occur < pages_copy[k])
    {
      max_occur = pages_copy[k];
      counter = k;
    }
  }
```

```
    max_occur = counter;
    pages[max_occur] = array_copy[final_counter];
    pages_copy[max_occur] = 1;
    page_faults++;
  }

  hit = (length+1)-page_faults;
  ratio = (hit*100)/(length+1);
  printf("\n\t\t\t\tMFU :\n\n");
  printf("No. of Hits : %2d\t",hit);
  printf("No. of Page Falut: %2d\t\t",page_faults);
  printf("Hit Ratio: %2d%%\n",ratio);
}

int main(int argc, char *argv[])
{
  int working_set_size = 0;
  char *line = NULL;
  size_t line_length = MAX_LINE;
  char *filename;

  FILE *file;

  if (argc < 2)
  {
    printf("Error: You must provide a checkfile as an argument.\n");
    printf("Example: ./fp checkfile.txt\n");
    exit(EXIT_FAILURE);
  }
```

```
filename = (char *)malloc(strlen(argv[1]) + 1);
line = (char *)malloc(MAX_LINE);

memset(filename, 0, strlen(argv[1] + 1));
strncpy(filename, argv[1], strlen(argv[1]));

printf("\nOpening file %s ...\n", filename);
file = fopen(filename, "r");

if (file)
{
  int pages[MAX_LINE];
  int length = 0;
  char copy_forMFU[MAX_LINE];
  char *line2 = (char *)malloc(MAX_LINE);

  while (fgets(line, line_length, file))
  {
    char *token;
    char *token2;

    strcpy(line2, line);
    token2 = strtok(line2, " ");
    token2 = strtok(NULL, "\r\n");
    strcpy(copy_forMFU, token2);

    token = strtok(line, " ");
    working_set_size = atoi(token);

    printf("\nNo. of Frame : %d\n\n", working_set_size);
```

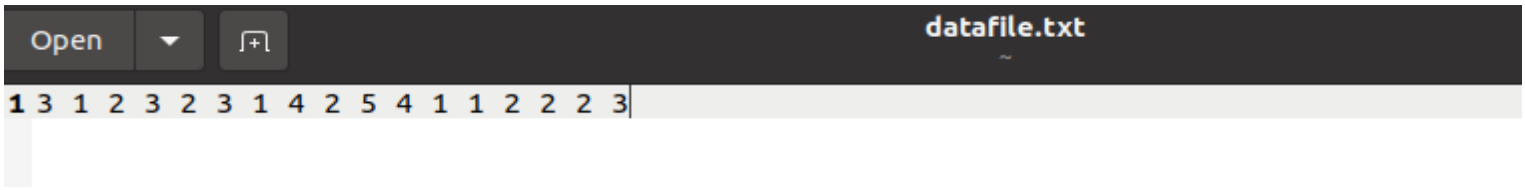
```
    while (token != NULL)
    {
      token = strtok(NULL, " ");
      if (token != NULL)
      {
        pages[length] = atoi(token);
        length++;
      }
    }

    FIFO_pagefault(pages, working_set_size, length);
    LRU_pagefault(pages, working_set_size, length);
    MFU_pagefault(working_set_size, copy_forMFU);
    optimal_pagefault(pages, working_set_size, length);
    length = 0;
    printf("\n");
  }
  free(line);
  fclose(file);
  printf("\nClosing file %s ...\n", filename);
 }
 else
 {
  perror("Couldnt open file :(  ....\n");
 }
 return 0;
}
```

COMMANDS

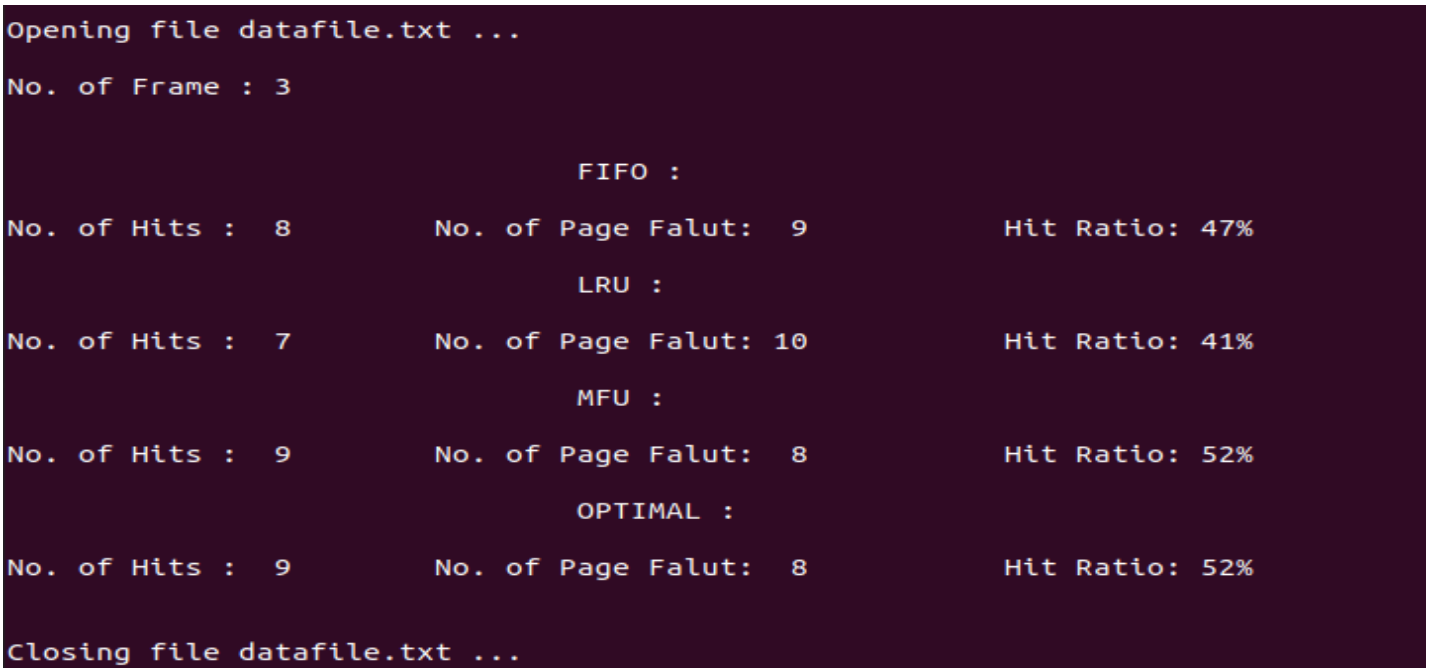


As seen on the screenshot above, we are editing our input file named as datafile.txt where we send our input, after compiling the code we are running the code by sending this txt file as an argument. After receiving this argument our code finds out the no of page faults and no of hits and hit ratio using the four algorithm studied in OS. The FIFO Algorithm, Optimal Algorithm, LRU Algorithm and MFU Algorithm. As you can see in the screenshots of output below we are solving the Paging problem.

OUTPUT



The above is our input file consisting of 18 numbers but will be passed as a string.



Above is the output of the data entered above showing no of hits, page faults and hit ratio.

The above is our input file consisting of 24 numbers but will be passed as a string

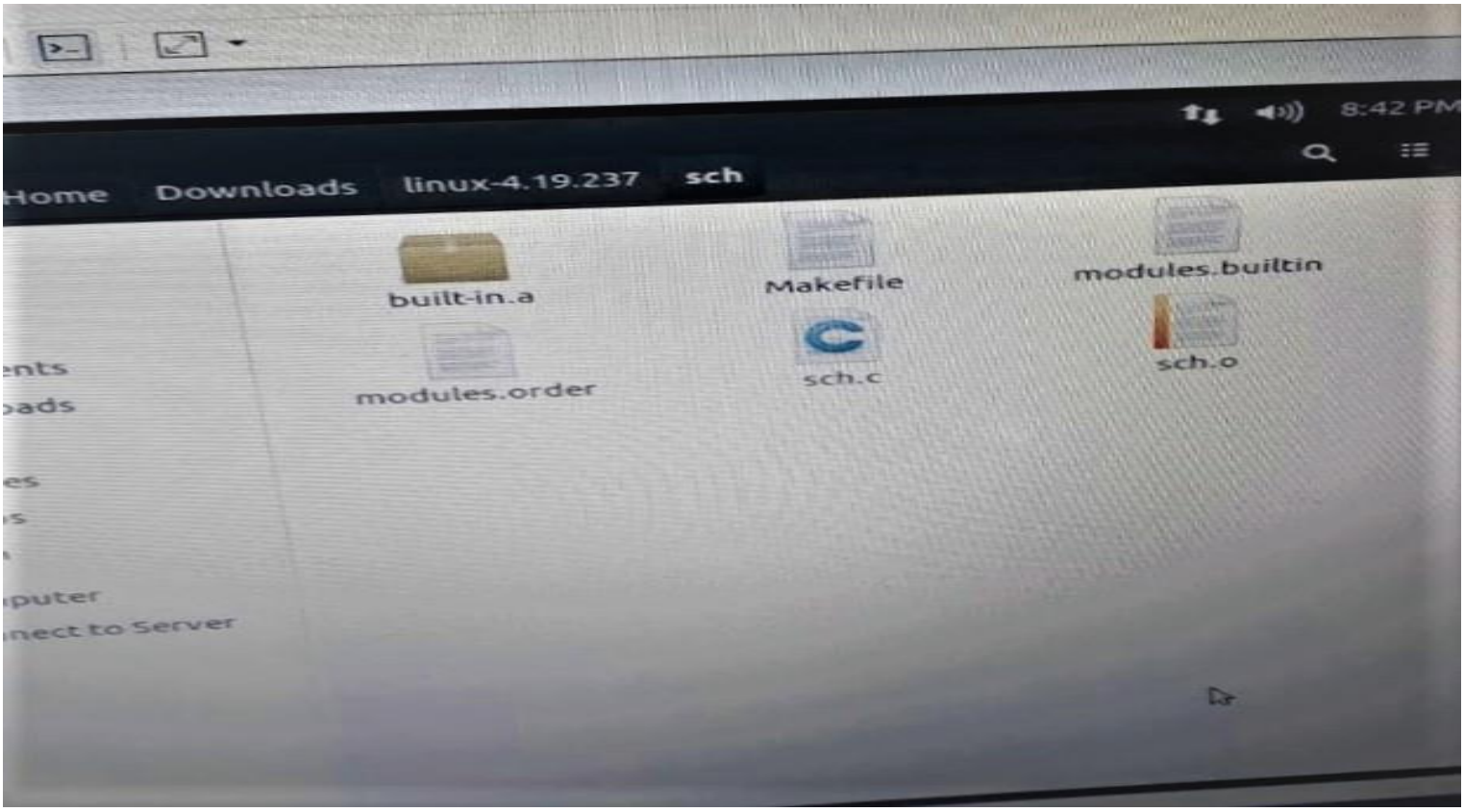
```
Opening file datafile.txt ...
No. of Frame : 5

                                FIFO :
No. of Hits : 13        No. of Page Falut: 10           Hit Ratio: 56%
                                LRU :
No. of Hits : 15        No. of Page Falut:  8           Hit Ratio: 65%
                                MFU :
No. of Hits : 14        No. of Page Falut:  9           Hit Ratio: 60%
                                OPTIMAL :
No. of Hits : 16        No. of Page Falut:  7           Hit Ratio: 69%

Closing file datafile.txt ...
```

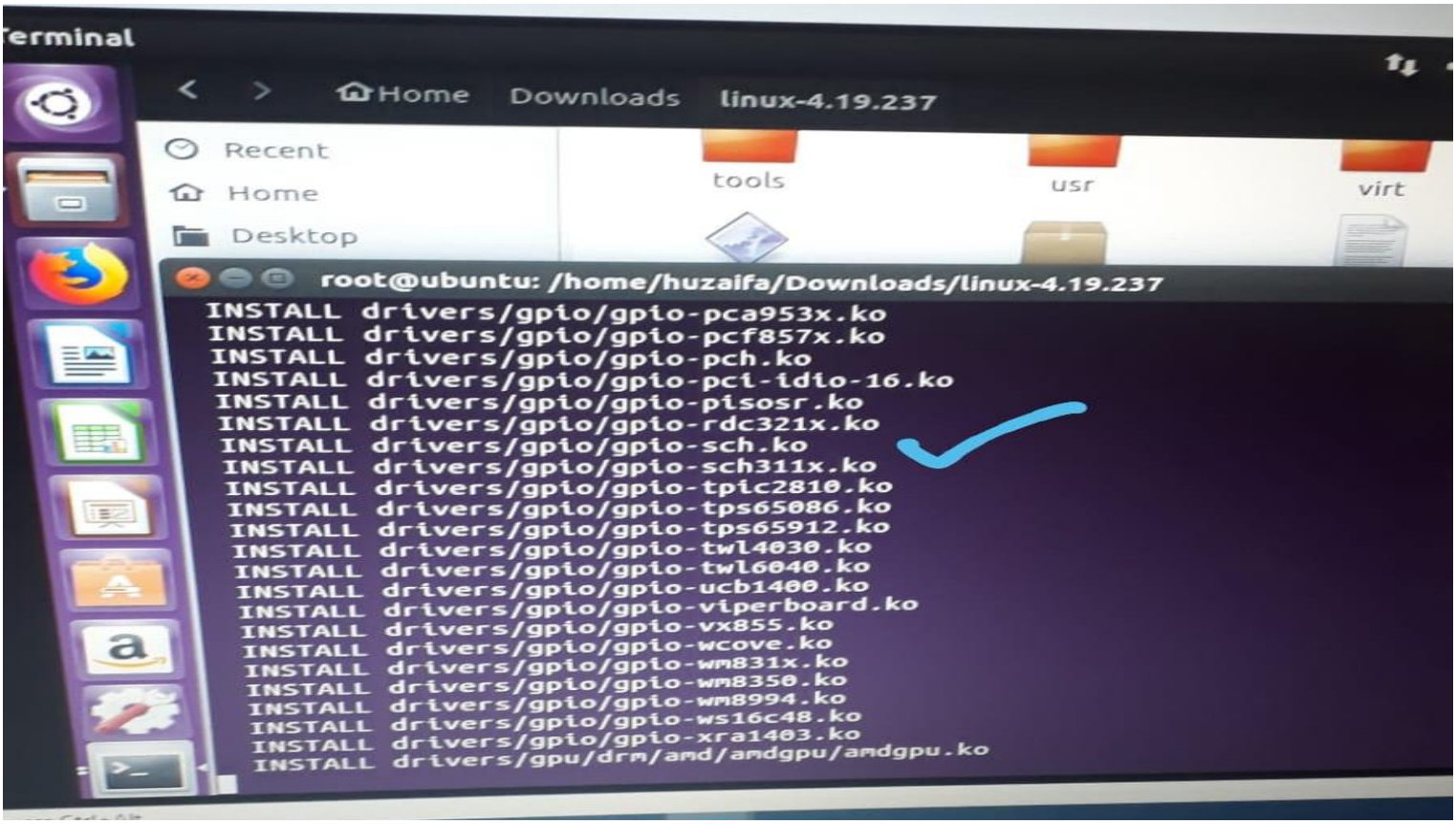
Above is the output of the data entered above showing no of hits, page faults and hit ratio. As you can see our optimal algorithms shows least no of page faults.

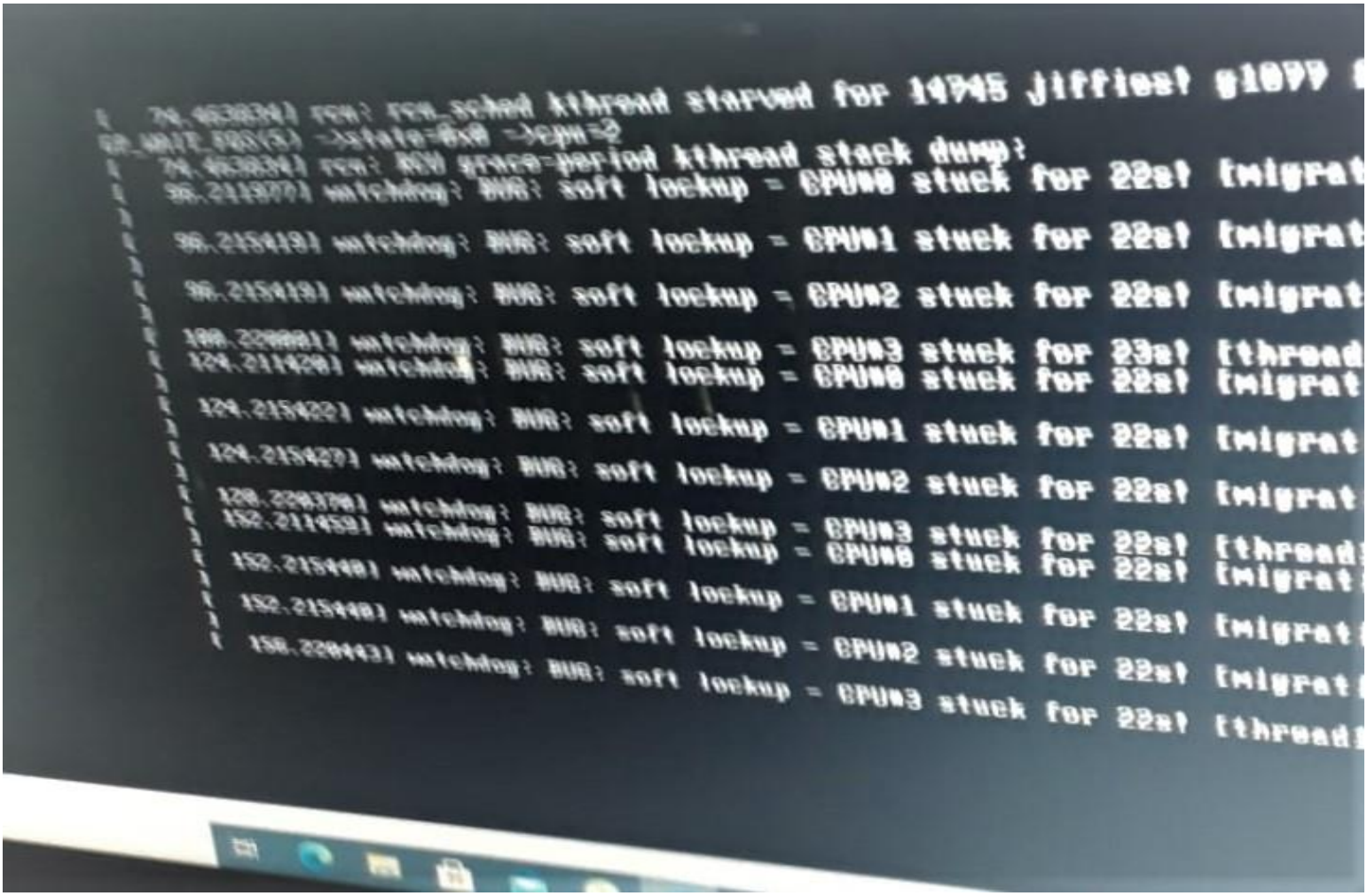
PROBLEMS FACED

- Passing string of references as an argument in a form of txt file and fetching out data from it to save our program from run time input where user can do mistakes and program can get stuck in infinite loop or show some abnormal behavior.
- First we decided to make a system call to run this on kernel mode, everything was going great , system call was compiled and modules were installed (I have uploaded screen shots of that progress),but unfortunately on very last system call was running fine but it does not shows desire output and just returning a value. We have spent lot of time to make it correct, almost three times we compiled our system call to just get work on kernel mode.
- At the same time as we knew that, that our program is bit simple, so we were working on deadlock, we were able to detect it, resolve it by terminating on deadlock causing process and also able to provide a safe state by using bankers algorithm(I have uploaded screen shots of that progress). Then we made a comparison among two programs and found replace algorithms more reliable and worthy with respect to marks. ”kindly consider our efforts”.

Working progress of System Call







Working progress of Deadlock Management

```
dehya@dehya: ~
dehya@dehya:~$ gedit osproject.c++
dehya@dehya:~$ g++ os osproject.c++
g++: error: os: No such file or directory
dehya@dehya:~$ g++ osproject.c++
dehya@dehya:~$ ./a.out
```

```
* Deadlock Detection Algo *

Process  Allocation       res      Available
P0          3 3 3         3 6 8    1 2 0
P1          2 0 3         4 3 3
P2          1 2 4         1 9 4
P3          0 0 1         9 1 6

System is in Deadlock and the Deadlock process are
P0       P1       P2       P3



Removing P0



Process  Allocation       res      Available
P1          2 0 3         4 3 3    4 5 3
P2          1 2 4         1 9 4
P3          0 0 1         9 1 6

No Deadlock Occur

Safe sequences are:
P1--> P2--> P3
P1--> P3--> P2
P2--> P1--> P3
P2--> P3--> P1
```

```
* Deadlock Detection Algo *


Process  Allocation       res      Available
P0          3 3 3         3 6 8    1 2 0
P1          2 0 3         4 3 3
P2          1 2 4         1 0 4
P3          0 0 1         2 1 1

No Deadlock Occur

Safe sequences are:
P1--> P2--> P0--> P3
P1--> P2--> P3--> P0
P1--> P3--> P2--> P0
P2--> P1--> P0--> P3
P2--> P1--> P3--> P0
P2--> P3--> P0--> P1
P2--> P3--> P1--> P0
P3--> P1--> P2--> P0
P3--> P2--> P0--> P1
P3--> P2--> P1--> P0
There are total 10 safe-sequences
```